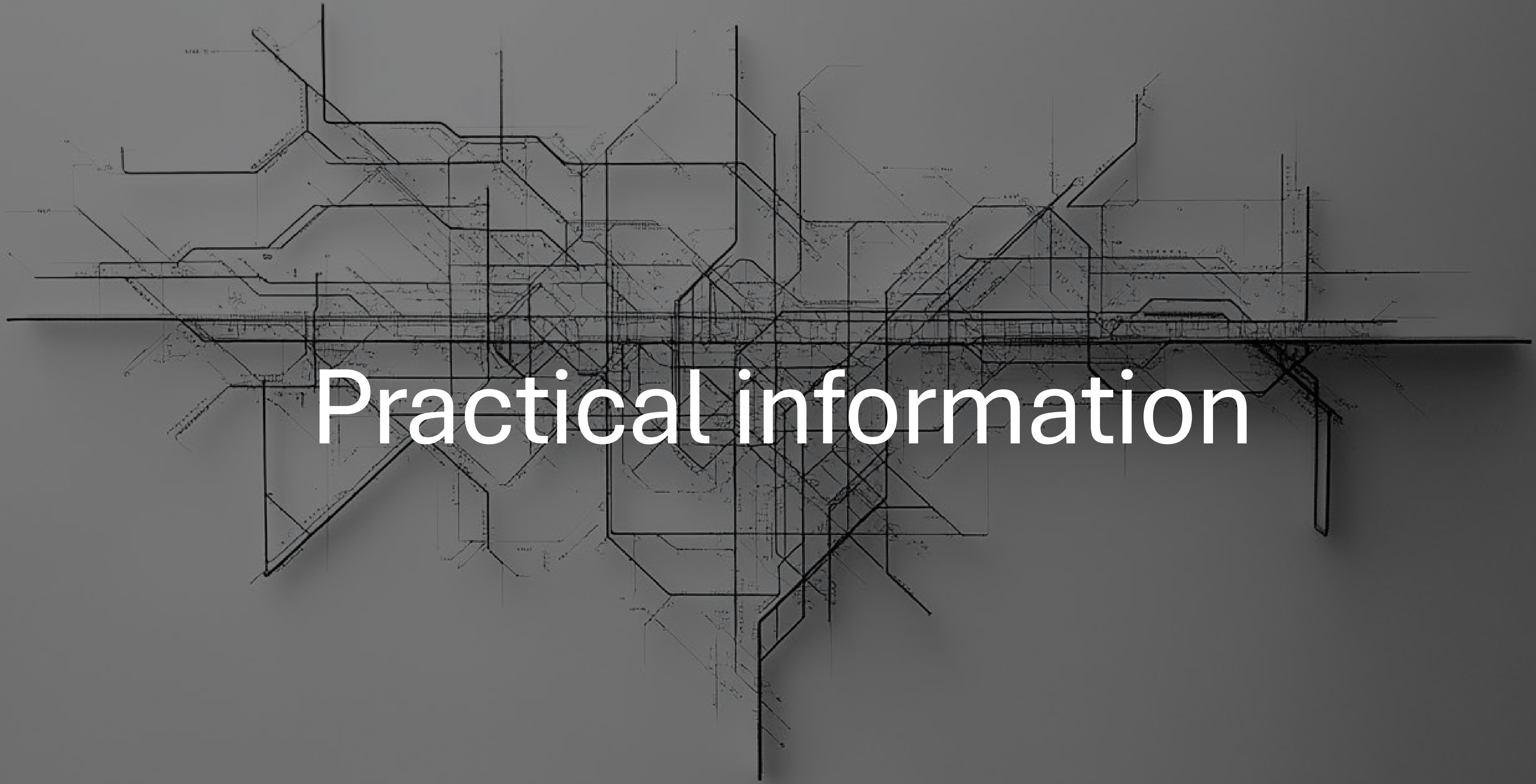
Array Computing with Eigen

Jonas Lindemann

jonas.lindemann@naiss.se

Jonas.Lindemann@lunarc.lu.se



Practical information

Schedule

Time	Descrption	Type
9.00 - 9.05	Introduciton	Lecture
9.05 - 9.15	Setting up development environment	Lab/Interactive
9.15 - 10.00	Working with matrices and vectors	Lecture
10.00 - 10.20	Coffee break	
10.20 - 11.00	Working with exercises	Lab
11.00 - 12.00	Advanced matrix operations	Lecture
12.00 - 13.00	Lunch	
13.00 - 13.40	Working with exercises	Lab
13.40 - 14.00	Best practice and integration	Lecture
14.00 - 14.30	Working with exercises	Lab
14.30 - 14.45	Coffee break	Lecture
14.45 - 15.15	Using Eigen in Parallel applications	Lecture

Course page

- Main page
  - https://array-computing-with-eigen.readthedocs.io/en/latest/index.html
- Exercises
  - https://array-computing-with-eigen.readthedocs.io/en/latest/exercises.html

Array Computing with Eigen

Show Source

This is a course on array computing with the C++ library Eigen. The course is intended for students with a basic knowledge of C++ programming and linear algebra.

Jonas Lindemann, 2025

Introduction

Why Not Build Your Own Matrix Library

- Arrays are crucial for scientific computing applications
- In C++, it might be tempting to create your own matrix library
- However, this is often not recommended because:
  - Existing libraries are well-tested
  - Existing libraries are highly optimized
  - Creating your own introduces unnecessary complexity

What is Eigen

- One of the most popular libraries for linear algebra in C++
- A header-only library (no linking required)
- Fast and easy to use
- Supports optimized packages like BLAS and LAPACK
- Can significantly speed up computational tasks
    - Some operations have support for OpenMP
    - Using optimized libraries will provide additional speed.

Setting up Eigen in Your Project

- Eigen is header-only, so no linking needed
  - Can still need linking for optimized BLAS / LAPACK libraries
- Simply include the relevant header:

```
#include <Eigen/Dense>
```

- The Dense module includes all basic matrix operations
- Most commonly used module for linear algebra tasks
- Other specialized modules are available for specific needs

Getting Started with Eigen

- Easy to integrate into existing C++ projects
- No external dependencies required
- Compatible with modern C++ standards
- Excellent documentation available
- Active community support

Setting Up Your Eigen Development Environment

Environment Setup

- COSMOS at LUNARC (HPC environment)
- Windows
- Linux (Ubuntu)
- macOS

Using COSMOS at LUNARC

- Remote desktop environment: LUNARC Documentation
- SSH access: SSH Login Documentation

LUNARC: Loading the Environment

Load required modules:

```
module load foss/2024a
module load Eigen
module load Cmake
```

Verify installation:

```
g++ --version      # Should show GCC 13.3.0
cmake --version    # Should show CMake 3.29.3
```

LUNARC: Testing Eigen

Create a test file `ex0.cpp`:

```cpp
#include <iostream>
#include <Eigen/Dense>

int main()
{
    Eigen::Matrix3d m = Eigen::Matrix3d::Random();
    std::cout << "Here is the matrix m:" << std::endl;
    std::cout << m << std::endl;
    return 0;
}
```

LUNARC: Compiling and Running

Compile:

```
g++ ex0.cpp -o ex0
```

Run:

```
./ex0
```

Expected output:

```
Here is the matrix m:
0.680375   0.59688 -0.329554
-0.211234  0.823295  0.536459
0.566198 -0.604897 -0.444451
```

Ubuntu Linux Setup

Install required packages:

```
sudo apt-get update
sudo apt-get install g++
sudo apt-get install cmake
sudo apt-get install libeigen3-dev
```

Verify installation:

```
g++ --version
cmake --version
```

Ubuntu Linux: Testing Eigen

Create the same test file `ex0.cpp` as before.

Compile with include path:

```
g++ ex0.cpp -I/usr/include/eigen3 -o ex0
```

Run:

```
./ex0
```

The output should match the previous example.

macOS Setup

Install required packages using Homebrew:

```
brew install gcc
brew install cmake
brew install eigen
```

macOS: Testing Eigen

Create the same test file `ex0.cpp` as before.

Compile with include path:

```
g++ -std=c++11 -I/usr/local/include/eigen3 ex0.cpp -o ex0
```

Run:

```
./ex0
```

The output should match the previous examples.

Troubleshooting Tips

- Check compiler version compatibility
- Verify include paths are correct
- Ensure Eigen headers are properly installed
- For library issues, confirm environment variables are set correctly

Demo

Working with Matrices and Vectors

Matrix and Vector Types

- All Eigen classes are template classes (work with different data types)
- Most common data types: **`float`**, **`double`**, and **`int`**
- All classes defined in the **`Eigen`** namespace
- Best practice: use **`Eigen::`** prefix instead of
  - **`using namespace Eigen`**
- In many of my example I will use without the prefix to make the code more readable.

Matrix Declaration

Generic form:

```
// 3x3 matrix of doubles
Eigen::Matrix<double, 3, 3> A;
```

Using convenient typedefs:

```
Eigen::Matrix3d B;  // Same as above
```

Vector Declaration

Generic form:

```
// 3x1 vector of doubles
Eigen::Vector<double, 3> v;
```

Using convenient typedefs:

```
Eigen::Vector3d w;  // Same as above
```

Note: Vectors are matrices with one dimension fixed to 1

Initialization

- Newly declared matrices contain **random values**
- Methods to initialize:
  - `A.setZero()` - Initialize to zeros
  - `A.setOnes()` - Initialize to ones
  - `<< operator` - Set specific values

```
Eigen::Matrix3d D;

D << 1, 2, 3,
     4, 5, 6,
     7, 8, 9;
```

Vectors - Special Properties

- Initialize with constructor:

```
Eigen::Vector3d v(1, 2, 3);
```

- Initialize with the << operator:

```
Eigen::Vector3d w;
w << 1, 2, 3;
```

- Special initialization methods:

```
v.setLinSpaced(3, 1, 2); // Creates: [1, 1.5, 2]
w.setRandom();           // Random values
```

Column and Row Access

```
// Column insertion

D.col(0) << 1, 2, 3;
D.col(1) << 4, 5, 6;
D.col(2) << 7, 8, 9;

// Row insertion

D.row(0) << 1, 2, 3;
D.row(1) << 4, 5, 6;
D.row(2) << 7, 8, 9;
```

Common Typedefs

Fixed-size matrices:

```
Matrix2d, Matrix3d, Matrix4d  // double
Matrix2f, Matrix3f, Matrix4f  // float
Matrix2i, Matrix3i, Matrix4i  // int
```

Dynamic-size matrices:

```
MatrixXd  // double
MatrixXf  // float
MatrixXi  // int
```

Common Vector Typedefs

Fixed-size vectors:

```
Vector2d, Vector3d, Vector4d  // double
Vector2f, Vector3f, Vector4f  // float
Vector2i, Vector3i, Vector4i  // int
```

Dynamic-size vectors:

```
VectorXd  // double
VectorXf  // float
VectorXi  // int
```

Demo

Fixed vs Dynamic Size

Fixed size (known at compile time):

```
// 3x3 matrix, size fixed at compile time

Eigen::Matrix3d A;
```

Dynamic size (determined at runtime):

```
Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic> F;
F.resize(3, 3);  // Set dimensions at runtime

// Shorthand

Eigen::MatrixXd G(3, 3);
```

Resizing Matrices

- `resize(rows, cols)` changes matrix dimensions
- If the total number of elements stays the same, data is preserved
- If the total elements change, data is lost and must be reinitialized

```
A_dyn.resize(1, 9); // Preserves data if A_dyn was 3x3
A_dyn.resize(6, 6); // Changes total elements, data is lost
```

Row Vectors

```
// 1x3 vector
Eigen::RowVector3d r(1.0, 2.0, 3.0);
// Output: 1 2 3

// 3x1 vector
Eigen::Vector3d s(1.0, 2.0, 3.0);
// Output:
// 1
// 2
// 3
```

Matrix Operations

- Addition: `A + B`
- Scalar multiplication: `A * 3.0`
- Matrix multiplication: `A * B`
- Add scalar to all elements: `E + Matrix3d::Constant(1.0)`
- Element-wise operations: `E.array() + 3.0`

Matrix Transformations

- Transpose: `A.transpose()`
- Inverse: `A.inverse()`
- Component-wise operations:

```
Vector3d x(1, 4, 9);
auto y = x.cwiseSqrt();  // [1, 2, 3]
// Or
auto w = z.array().sqrt();
```

Vector Operations

- Dot product: `s.dot(t)`
- Cross product: `s.cross(t)`
- Component-wise operations: `x.cwiseSqrt()`

Reduction Operations

- `K.sum()` - Sum of all elements
- `K.prod()` - Product of all elements
- `K.mean()` - Mean of all elements
- `K.norm()` - Euclidean norm
- `K.maxCoeff()` - Maximum value
- `K.minCoeff()` - Minimum value
- `K.trace()` - Trace of the matrix
- `K.diagonal()` - Diagonal elements
- `K.determinant()` - Determinant

Demo

Coffee break

10:00 – 10:20

Exercise

10:20 – 11:00

Advanced Matrix Operations in Eigen

Reshaping Matrices

- `.reshaped(rows, cols)` method allows changing matrix dimensions
- Returns a view into the original matrix (not a copy)
- Changes to the reshaped matrix affect the original

```
Matrix3d A;

A << 1, 2, 3,
     4, 5, 6,
     7, 8, 9;

auto B = A.reshaped(1, 9); // Result: 1 2 3 4 5 6 7 8 9
```

Reshaping Considerations

- Data is stored in column-major order in Eigen
- Self-assignment requires `.eval()` to force evaluation:

```
C = C.reshaped(1, 9).eval();
```

- Can combine with `.transpose()`:

```
MatrixXd D = C.reshaped(1, 9).transpose();
```

Demo

Slicing and Indexing

Row and column access:

```
// Set values in row 3
A.row(3) << 1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

// Set values in column 3
A.col(3) << 1, 2, 3, 4, 5, 6, 7, 8, 9, 10;

// Set all elements in column 1 to 1
A.col(1).setOnes();
```

Range Selection with `seq()`

Select ranges of rows and columns:

```
// Select block from rows 3-5, columns 3-5
B(seq(3, 5), seq(3, 5)).setConstant(1);

// Select every other row/column from 0-9
B(seq(0, 9, 2), seq(0, 9, 2)).setConstant(2);
```

Special Selectors

```
// Set all elements in the last column to 3
B(all, last).setConstant(3);

// Set all elements in the second-to-last
// column to 4
B(all, last - 1).setConstant(4);
```

Index-Based Selection

Use `std::vector` of indices to select specific rows and columns:

```
vector<int> idx = { 1, 3, 4, 6, 7, 9 };

// Select submatrix using index vector
auto D = C(idx, idx);
```


Demo

Linear System Solving - Small Matrices

For small matrices (up to 4x4), using inverse is acceptable:

```
Matrix3d A;
A.setRandom();
Vector3d b;
b.setRandom();

// Solve Ax = b
Vector3d x = A.inverse() * b;
```

Linear System Solving - Larger Matrices

For larger matrices, use decomposition methods:

```
MatrixXd A(10, 10);
A.setRandom();
VectorXd b(10);
b.setRandom();

// Solve using QR decomposition
VectorXd x = A.colPivHouseholderQr().solve(b);

// Check error
double error = (A * x - b).norm();
```


Demo

Matrix Decompositions

Different decompositions for different matrix types:

- `colPivHouseholderQr()` - General matrices (robust)
- `fullPivLu()` - General matrices (most stable, slower)
- `ldlt()` - Symmetric matrices
- `householderQr()` - General matrices (fastest, less accurate)

Reusing Decompositions

Create decomposition objects for reuse with multiple right-hand sides:

```
// Create decomposition once
FullPivLU<MatrixXd> lu(A);

// Solve for multiple right-hand sides
VectorXd x1 = lu.solve(b1);
VectorXd x2 = lu.solve(b2);

// Solve systems with multiple right-hand sides at once
MatrixXd X = lu.solve(B);  // B has multiple columns
```

Demo

Lunch

12:00 – 13:00

Eigen Best Practices and Integration

Returning Matrices from Functions

- Prefer returning Eigen matrices by value
- C++ return value optimization (RVO) prevents unnecessary copying
- Simple and effective approach:

```
MatrixXd foo()
{
    MatrixXd A(10, 10);
    A.setRandom();
    return A;  // Efficient due to RVO
}

int main()
{
    MatrixXd B = foo();  // No unnecessary copying
}
```

Passing Matrices to Functions

- Pass matrices as const references to avoid copying
- Use the `const` keyword to indicate the matrix won't be modified

```
void bar(const MatrixXd& A)  // Pass by reference
{
    cout << A << endl;
}

int main()
{
    MatrixXd B(10, 10);
    B.setRandom();
    bar(B);  // No copying occurs
}
```

Implementing Functions with Eigen

General Rule: - Pass Eigen matrices and vectors by reference - Return Eigen matrices and vectors by value

```
// Example: Function that creates and returns a matrix
MatrixXd hooke(TAnalysisType ptype, double E, double v)
{
    MatrixXd D;
    // ... matrix construction ...
    return D;  // Return by value
}
```

Example: Function Implementation

```
Matrix4d bar2e(const Vector2d& ex, const Vector2d& ey, const
Vector2d& ep)
{
    // Parameters passed by const reference
    double E = ep(0);
    double A = ep(1);
    double L = sqrt(pow(ex(1)-ex(0),2)+pow(ey(1)-ey(0),2));

    // ... calculations ...

    Matrix4d Ke = G.transpose()*Ke_loc*G;
    return Ke;  // Return by value
}
```

Accessing Raw Data

- Use `.data()` method to get pointer to raw data
- Useful for interfacing with C-style APIs

```
MatrixXd A(10, 10);
A.setRandom();

// Get pointer to underlying data
double* data = A.data();

// Access elements directly
for (int i = 0; i < A.size(); i++) {
    cout << data[i] << " ";
}
```

Creating 2D Array Views

- Eigen stores data as 1D array internally
- Create array of pointers for 2D access:

```cpp
double* data = A.data();
double** data2D = new double*[A.rows()];

// Setup row pointers
for (int i = 0; i < A.rows(); i++)
    data2D[i] = data + i * A.cols();

// Clean up (only deletes pointer array, not actual data)
delete[] data2D;
```

Integrating with Other Libraries

- When working with libraries requiring 2D C-style arrays:

```cpp
void foo(double** data, int rows, int cols) {
    // Function expecting 2D C-style array
}

// Setup for integration
double** data2D = new double*[A.rows()];
for (int i = 0; i < A.rows(); i++)
    data2D[i] = A.row(i).data();

// Call external function
foo(data2D, A.rows(), A.cols());

// Clean up
delete[] data2D;
```

Dealing with Const Issues

- Use `const_cast` when necessary for C-style API integration:

```
// If compiler warns about const correctness
data2D[i] = const_cast<double*>(A.row(i).data());
```

- Only do this when sure the data won't be modified
- Avoid if possible - const correctness is a feature, not a bug

Summary of Best Practices

1. Return matrices by value
2. Pass matrices by const reference
3. Use `.data()` for raw data access
4. Create proper interfaces for C-style libraries
5. Maintain const correctness when possible
6. Be aware of memory ownership when interfacing

Demo

Exercise

10:20 – 11:00

Coffee break

10:00 – 10:20

Using Eigen in Parallel Applications

Overview

- Eigen provides efficient array and vector data types
- Takes advantage of underlying linear algebra libraries if available
- Raw data accessible through `.data()` method
- Many operations support OpenMP parallelization

Eigen and OpenMP Support

Eigen operations that use OpenMP for parallelization:

- general dense matrix - matrix products
- PartialPivLU
- row-major-sparse * dense vector/matrix products
- ConjugateGradient with Lower|Upper as the UpLo template parameter.
- BiCGSTAB with a row-major sparse matrix format.
- LeastSquaresConjugateGradient

Using Eigen with OpenMP

Simple example: Matrix multiplication using Eigen’s built-in OpenMP support

```
// Setting number of threads
omp_set_num_threads(n_threads);

// This operation will automatically use OpenMP
MatrixXd C = A * B;
```

OpenMP Performance Scaling

Performance scaling for matrix multiplication (20000×20000):

Threads	Time (seconds)
1	870.111
2	427.386
4	217.594
8	109.546
12	74.313
24	40.574
48	22.657

Custom Parallelization (Part 1)

Using Eigen for storage with custom OpenMP implementation:

```cpp
// Use RowMajor for more efficient row operations
using Matrix = Eigen::Matrix<double, Eigen::Dynamic,
                             Eigen::Dynamic, Eigen::RowMajor>;
using Vector = Eigen::VectorXd;

Vector customMatVecMult(const Matrix& A, const Vector& x) {
    const int rows = A.rows();
    const int cols = A.cols();
    Vector result(rows);
```

Custom Parallelization (Part 2)

```
    // Get raw pointers to the data
    const double* A_data = A.data();
    const double* x_data = x.data();
    double* result_data = result.data();

    #pragma omp parallel for schedule(static)
    for (int i = 0; i < rows; i++) {
        double sum = 0.0;
        const double* row = A_data + i * cols;

        for (int j = 0; j < cols; j++) {
            sum += row[j] * x_data[j];
        }

        result_data[i] = sum;
    }

    return result;
}
```

Performance Comparison

Custom OpenMP implementation vs. Eigen's built-in (40000×40000):

```
Matrix size: 40000x40000
OpenMP implementation time: 352ms
Eigen implementation time: 389ms
Relative error: 7.35186e-15
```

Custom implementation slightly outperforms Eigen's built-in operation.

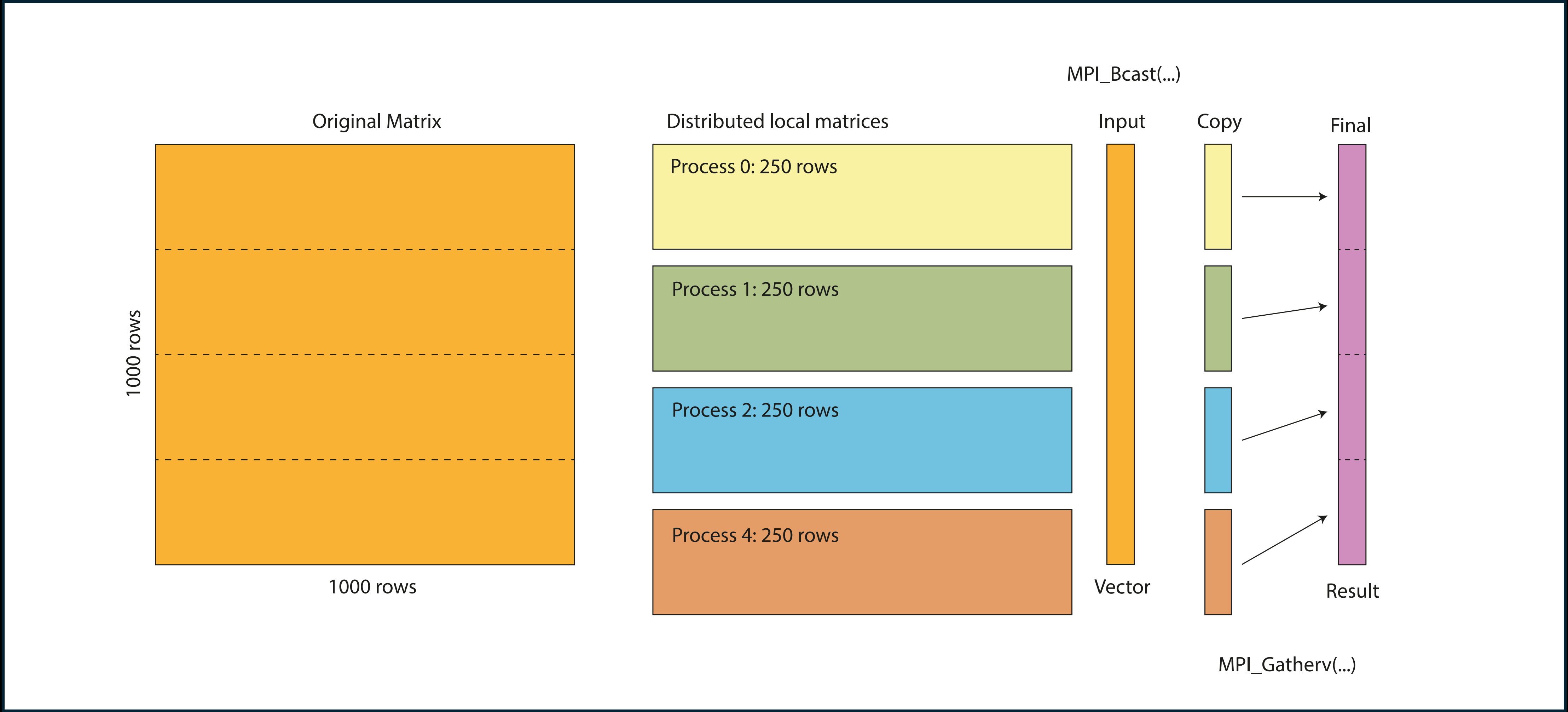
Using Eigen with MPI

- Eigen is NOT a distributed Matrix library
- MPI for distributed memory parallelism
- Eigen arrays stored in 1D memory layout
- Developer handles data distribution
- Example: matrix-vector multiplication across multiple processes

MPI Distribution Strategy

- Matrix divided by rows
- Each process handles a portion of rows
- Vector broadcasted to all processes
- Each process computes partial result

MPI Distribution Strategy



MPIMatrix Class (Part 1)

```cpp
class MPIMatrix {
private:
    int m_rank;
    int m_size;
    int m_rows;
    int m_cols;
    MatrixXd m_localMatrix;
    VectorXd m_localResult;

public:
    MPIMatrix(int r, int c)
        : m_rank(0), m_size(1), m_rows(r), m_cols(c)
    {
        MPI_Comm_rank(MPI_COMM_WORLD, &m_rank);
        MPI_Comm_size(MPI_COMM_WORLD, &m_size);
```

MPIMatrix Class (Part 2)

```
        // Calculate local matrix size
        int localRows = m_rows / m_size;
        if (m_rank < m_rows % m_size) {
            localRows++;
        }

        m_localMatrix.resize(localRows, m_cols);
        m_localResult.resize(localRows);
    }

    void randomize() {
        
srand(std::chrono::system_clock::now().time_since_epoch().count());
        m_localMatrix.setRandom();
    }
```

MPIMatrix Class (Part 3)

```cpp
void multiply(const VectorXd& vec) {
    // Local multiplication
    m_localResult = m_localMatrix * vec;
}

void printResult() const {
    // Gather results
    std::vector<int> recvCounts(m_size);
    std::vector<int> displs(m_size);

    // Calculate receive counts and displacements
    int baseCount = m_rows / m_size;
    int remainder = m_rows % m_size;
```

MPIMatrix Class (Part 4)

```
for (int i = 0; i < m_size; ++i) {
    recvCounts[i] = baseCount + (i < remainder ? 1 : 0);
    displs[i] = (i > 0) ? displs[i-1] + recvCounts[i-1] : 0;
}

// Allocate space for complete result
VectorXd globalResult;
if (m_rank == 0)
    globalResult.resize(m_rows);

// Gather all local results to rank 0
MPI_Gatherv(m_localResult.data(), m_localResult.size(), MPI_DOUBLE,
            globalResult.data(), recvCounts.data(), displs.data(),
            MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

MPIMatrix Class (Part 5)

```
        // Print result on rank 0
        if (m_rank == 0) {
            std::cout << "First few elements of result: \n"
                      << globalResult.head(5).transpose() << std::endl;
        }
    }
};
```

MPI Main Function (Part 1)

```
int main(int argc, char** argv) {
    constexpr int MatrixSize = 10000;
    int rank;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Create distributed matrix
    MPIMatrix distMatrix(MatrixSize, MatrixSize);
    distMatrix.randomize();
```

MPI Main Function (Part 2)

```
// Create a random x vector
VectorXd x;
if (rank == 0) {
    std::cout << "Generating random vector x...\n";
    x = VectorXd::Random(MatrixSize);
} else {
    x.resize(MatrixSize);
}

// Broadcast x vector to all processes
MPI_Bcast(x.data(), MatrixSize, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

MPI Main Function (Part 3)

```
// Perform distributed matrix-vector multiplication
auto startTime = std::chrono::high_resolution_clock::now();
distMatrix.multiply(x);
auto endTime = std::chrono::high_resolution_clock::now();

if (rank == 0) {
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime-startTime);
    std::cout << "Matrix size: " << MatrixSize << " x " << MatrixSize << std::endl;
    std::cout << "Matrix memory size (MB): "
              << sizeof(double) * MatrixSize * MatrixSize / 1e6 << std::endl;
    std::cout << "Multiplication completed in " << duration.count() << " ms\n";
}

distMatrix.printResult();
```

MPI Considerations

- For large datasets, avoid gathering all data to one process
- Consider:
  - Writing results to separate files on each rank
  - Using parallel I/O libraries like HDF5
  - Using MPI I/O functions

Summary: Eigen in Parallel Applications

- Eigen works well with OpenMP and MPI
- Use built-in Eigen parallel operations when possible
- For custom parallelization:
  - Access raw data with `.data()`
  - Use RowMajor order for row-wise operations
  - Implement your own parallelization with OpenMP/MPI
- Benefits:
  - Simplified memory management
  - Clean, maintainable code
  - Good performance


Questions / Answers